Software Engineering Institute

A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications

Richard P. Kendall
Douglass E. Post
Jeffrey C. Carver
Dale B. Henderson
David A. Fisher

January 2007

TECHNICAL NOTE
CMU/SEI-2006-TN-039

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

Abstract

Because the development of large-scale scientific/engineering application codes is an often difficult, complicated, and sometimes uncertain process, success depends on identifying and managing risk. One of the drivers of the evolution of software engineering, as a discipline, has been the desire to identify reliable, quantifiable ways to manage software development risks. The taxonomy that follows represents an attempt to organize the sources of software development risk for scientific/engineering applications around three principal aspects of the software development activity: the software development cycle, the development environment, and the programmatic environment. These taxonomic classes are divided into *elements* and each element is further characterized by its *attributes*.

Introduction

Because the development of large-scale scientific/engineering application codes is an often difficult, complicated, and sometimes uncertain process, success depends on identifying and managing risk. Failure of some sort has been a common occurrence in the software development milieu; the literature indicates that more than 25% of all software development projects are cancelled outright before completion and something like 80% overrun their budgets. It is no surprise, then, that one of the drivers of the evolution of software engineering, as a discipline, has been the desire to identify reliable, quantifiable ways to manage software development risks (the possibility of suffering harm or loss, or "the product of uncertainty associated with project risks times some measure of the magnitude of the consequences" [Schmidt 2001]), stemming from, for example

- uncertain or inaccurate requirements
- requirements that change too rapidly
- overly optimistic scheduling
- institutional turmoil, including too much employee turnover
- poor team performance [DeMarco 1999]

Of course, these are risks for software development projects of every stripe, not just scientific/engineering applications. Some important attributes specific to scientific/engineering software application development projects that impact risk include the following:

- Although the laws of nature, the ultimate source of requirements for scientific/engineering applications, are fixed, they may not be well understood for a given application. For example, there is the issue of emergent behavior of physical phenomena that cannot be anticipated at the start of a project. Moreover, a given set of physical requirements may be represented by different, equally valid, but possibly mutually inconsistent algorithms in multi-physics applications.
- Scientific and engineering code development is often very lengthy—spanning decades and careers, so that the target problems of interest and the needs of the users necessarily evolve and change during code development.
- To some extent it is true of most such projects that the requirements at levels below the most obvious are not specified by the sponsor or customer because of (1) a desire to allow the project some flexibility in execution, and (2) the fact that lower level details are not known or even understood by the sponsor or customer. General project goals may be specified by the sponsor or customer, but these stakeholders often rely on the software development team itself, or its management, to translate these very-high-level goals into requirements.
- Scientists tend to be averse to complicated or expensive "process"-oriented software development methodologies whose value they question.
- The principal driver of scientific/engineering applications is, not surprisingly, science and engineering; the developers of these codes are often not explicitly funded or trained to do software engineering.

- Any scientific/engineering code running on a finite state machine—a computer—can only *approximate* the generally continuous laws of nature. There are inherent inaccuracies that arise from representing physical entities as object-based models.
- Many scientific/engineering software application development projects start out as research projects, and the codes as research codes. In this state there cannot be a definitive relationship between deliverables, schedule, and resources.
- It is often true that the developers are also the users or a significant part of the user community.
- Scientific and engineering code development is strongly driven by prototypes, for example prior codes that provide simpler, usually less comprehensive simulations of the laws of nature. In a sense, science is a model of nature that scientists are continually improving and refining. Scientific/engineering software is usually the latest embodiment of scientific models.

The taxonomy that follows represents an attempt to organize the sources of software development risk for scientific/engineering applications around three principal aspects of the software development activity:

- the software development cycle
- the development environment
- the programmatic environment

These taxonomic classes are divided into *elements* and each element is further characterized by its *attributes*.

Table 1: A Taxonomy for Sources of Software Development Risk in Scientific/Engineering Applications

A. Development Cycle Risks	B. Development Environment Risks	C. Programmatic Risks
1. Requirements Risks a. Predictability b. Evolvability c. Completeness d. Clarity e. Accuracy f. Precedence g. Execution Performance Expectations h. Proportionality 2. Design Risks a. Difficulty b. Modularity c. Usability d. Maintainability e. Portability f. Reliability 3. Implementation Risks a. Specifications b. Project Plan c. Scale of Effort 4. Test and Evaluation Risks a. Verification i. Unit Testing ii. Integration Testing iii. Interoperability Testing b. Validation	1. Development Process Risks a. Repeatability b. Suitability c. Control of Process d. Familiarity with Process or Practice e. Environment Change Control 2. Development System Risks a. Hardware Capacity b. Development System Capability c. Suitability d. Usability e. Familiarity f. Reliability g. Target-Unique System Support h. Security 3. Management Process Risks a. Contingency Planning b. Project Organization c. Management Experience d. Program Interfaces e. Reward Systems 4. Management Methods Risks a. Monitoring b. Personnel Manage-ment (Staffing and Training) c. Quality Assurance d. Configuration Management 5. Work Environment Risks a. Quality Attitude b. Cooperation c. Communication d. Morale e. Trust	1. Resources Risks a. Schedule b. Staff c. Budget d. Facilities e. Management Commitment 2. Contract Risks a. Contract Type b. Restrictions c. Dependencies 3. Program Interface Risks a. Customer Communication b. User Commitment c. Sponsor Alignment d. Subcontractor Alignment e. Prime Contractor f. Corporate Communication g. Vendor Performance h. Political

A. Development Cycle Risks

The development cycle encompasses the activities that are associated with the development of production-worthy code: requirements gathering, code design, the formulation of specifications, project planning, implementation, and testing.

The elements/attributes associated with this class of risks will be limited in this taxonomy to those sources of risk associated with the deliverable itself, independent of sources of risk associated with the *processes* or *tools* used to produce it or programmatic risks introduced by finite resources or external factors beyond project control. The risks arising from this class are usually considered to be intrinsic risks, that is, risks that are manageable from within the software development project itself. Risks that derive from external constraints are usually extrinsic, and those associated with processes or tools lie somewhere in between.

1. Requirements Risks

Requirements analysis, definition and management are intrinsic elements of life-cycle management. Risk attributes of the requirements risk element are associated with both the *quality* of the software requirements specification and also the *difficulty* of implementing software that satisfies the requirements.

In projects that start from poorly articulated requirements, as scientific/engineering projects often do, there is inherently far more risk that imprecisely expressed expectations will not be met. Technically difficult or imprecise requirements, coupled with the inability to negotiate relaxed requirements or budgets or schedules is a well-recognized source of software engineering risk.

The following attributes will be employed to illuminate the nature of the risks that are associated with the "requirements" element.

a. Predictability

Unexpected changes in requirements—that is, "unpredictability"—has been cited as the *sixth* highest source of risk that all software projects face [Keil 1998].

Within the scientific/engineering software development environment, a lack of predictability in requirements is often a consequence of the evolutionary nature of the requirements themselves. Many scientific/engineering code development projects begin as research projects or have research components throughout their life cycles. As such there is an inherent unpredictability about the requirements that must be addressed in the budgets and schedules of the project. Often the "iron triangle" of requirements, budgets, and schedules is inverted, with budgets and schedules fixed and requirements the only unconstrained variable. Even so, change control remains an important risk management tool.

This attribute also covers the risk associated with an evolving technical architecture.

b. Evolvability

The failure to recognize and adequately address the continuous evolution of requirements, that is "evolvability," is an especially important source of risk in long-lived scientific and engineering projects. This is particularly true in the period before the first significant deliverable, which may be the better part of a decade for a complex multi-physics code. Such projects sometimes fail because they cannot manage evolving requirements.

c. Completeness

Incomplete requirements fail to describe either the full intent or true intent (or both) of the customer. The principal consequence of this source of risk is that scope cannot be aligned with schedule and budget (resources). One purpose of the specification step (below) is to better codify the requirements and the intended design so that schedule and budget issues may be more fully addressed. Moreover, testing and code verification, also described below, cannot be very rigorous or complete without requirements against which to test.

d. Clarity

Clarity here is synonymous with understandability. Understandability is especially important when high-level goals are expressed by a customer who expects the developers to translate them into actionable requirements or a complete specification. The consequence of the lack of "clarity" is that the true intent of the requirements may not be discovered until it impacts negatively the schedule or budget of the project.

Lack of clarity of the requirements has been cited as the *third* highest source of risk faced by all software development projects [Schmidt 2001, Keil 1998].

e. Accuracy

Accuracy refers to the expectation that the aggregate requirements, which are likely to evolve in the case of many scientific/engineering software projects, reflect customer intentions or expectations for the application. If the requirements do not capture customer expectations, customer commitment to the project may be jeopardized. User commitment, which typically depends strongly on validity of the requirements, has been cited as the *second* highest source of software risk [Schmidt 2001, Keil 1998].

f. Precedence

Any software development project that posits capabilities that have not been demonstrated in existing software or that are beyond the experience of the project team or institution—that is, for which there is no "precedent"—may be vulnerable to this source of risk. The consequence may be that the project managers and team may not recognize that the objectives are infeasible.

g. Execution Performance Expectations

If execution performance is a major driver of the code development project, then these expectations must be addressed in the requirements, design, specifications, and testing of the application. Sometimes prototypes offer the best way to determine if performance expectations can be at-

tained. Too little attention to this source of risk usually results in a discarded code that is otherwise a technical success.

Especially in the high-performance computing (HPC) environment of cutting-edge hardware, there exists the possibility that tacit and/or written performance expectations cannot be met. Examples of performance metrics sometimes specified in contracts include:

- degree of performance optimization
- degree of parallel scaling
- fraction of theoretical peak performance for specified benchmarks
- job processing time for important applications of the code
- number of production runs per unit time (week, month, etc.)

Execution performance expectations sometimes drive optimization to the detriment of the accuracy, flexibility and utility of the resulting code.

h. Proportionality

Proportionality refers to the possibility that the requirements may be disproportionate to the solution, that is, that the problem is over-specified. For example, too many and too specific nonessential requirements can preclude feasible solutions. This source of risk is not confined to technical requirements; they often enter through management mandates that impact the function of the development team.

2. Design Risks

Design encompasses those steps through which requirements are translated into an actionable development plan. We distinguish three steps: software architecture (abstract or conceptual design), specification, and design per se. The software architecture should be influenced by the scientific domain and the mathematical attributes of the application (e.g., initial value problem, steady-state problem, eigenvalue problem, time evolution problem). With only the requirements and architecture, many software implementations are admitted; with a complete design and complete specification, there are far fewer options. It is important therefore to vet these documents (usually referred to as a "baseline") both ways: with the sponsor/customer to ensure that requirements (even unstated) and expectations will be met, and with the implementation team to ensure that they are confident of successful implementation. Finally, it is important that specifications and design be documented and kept up-to-date (this is referred to as "baseline management"); otherwise the work breakdown structure, scheduling, and budgeting (which should be based upon them) will be faulty. This is typically assessed in a critical design review (CDR).

Another sometimes overlooked design risk is the impact of design on testing. Difficulty in testing may begin with failure to include test features, especially those important to users, in the design.

The following attributes characterize different aspects of the risks inherent in the design element. It is also important to recognize that while documentation is very important, there is a risk of becoming too enamored with this aspect of project management at the expense of continuous validation against changing needs.

a. Difficulty

The existence of functional or performance requirements or expectations that are believed at the outset to be "difficult" should be viewed as a potential source of risk. The obvious consequence is that these requirements may not be met. Scientific/engineering software systems must deal with complications like algorithm adequacy, mesh generation, problem setup, very complex debugging and visualization, and the shear size of the calculations. The most serious manifestation of this complexity may be that the requirements are discovered to be infeasible, that is, in conflict. Often a major project should be preceded by a serious feasibility study; the results of which may cause redefinition of requirements or even abandonment of the project before any coding is begun.

b. Modularity

Modularity refers to the extent to which the code has been created using components or units that function independently from the other components. Software that has many direct interrelationships between different parts of the code is said to be less modular. Ideally, scientific code should be developed in modules that interact through well-defined interfaces and that capture functionality that can be independently tested (verified and validated). Generally speaking, the less modular a code is, the more likely it will not be maintainable or portable.

c. Usability

Usability is in the eye of the beholder, and since it is often true that the developers of scientific/engineering codes are often users, this aspect of the requirements receives too little attention. Even for developers, code that is difficult to use can strongly penalize the production phase and even the testing phase of a software project. In scientific codes, a lack of usability often manifests as opaque configuration options or grids that are difficult to set up. Scientific code developers, while they may be experts in their scientific domains, may not be experts in usability, human factors, or even the use of their own codes by others. Usability is emerging as a major risk within the HPC development environment as multi-discipline applications become ever more complex.

d. Maintainability

Maintainability, the ability to support the production phase of the product's life cycle with a reasonable level of resources, may be placed at risk by poor software architecture, design, specification, coding, or documentation. These failures may result from undefined or un-enforced standards, or from neglecting to analyze the system from the perspective of future maintainability. Maintainability is a strong function of attributes such as

- module cohesiveness and exclusivity
- interface minimization
- avoidance of complexity (the "KISS" principle)
- transparency and coding style

Because many significant scientific/engineering applications spend decades in the production/maintenance stage of the software life cycle, risks associated with a lack of maintainability must be addressed in design and implementation. The main consequence of a lack of maintain-

ability is a shortened life expectancy of the code. Some typical measures or indicators of maintainability include [Process Impact]

- lines of code (the most common measure but least indicative of maintainability)
- function points
- cyclomatic or Halstead complexity
- data coupling
- comment lines
- modularity (cited above)
- level of documentation (including programmer documentation)
- logical flow

e. Portability

While some scientific/engineering codes are used only once, or only with one type of computer, the majority of these codes outlast the generation of computers that they are first installed on, or are required to run on multiple hardware platforms from the beginning. Portability is often a necessity when developing code for HPC environments, because access to the target platform may be too limited to support the development timelines. Consequently, codes must be designed with portability as a conscious goal. The consequences of inattention to this issue range from unplanned effort to migrate the code to new hardware to, in the worst case, a complete rewrite. Codes that are not portable have short lives.

There is a strong link between portability and maintainability. The evolution of computer architectures is driven by the need to increase performance, not the need to promote portability. Ideally, the architectures of scientific/engineering software should be sufficiently flexible so that the software can be ported to future generations of computers with architectures that are currently unknown. To the extent that advances in performance are achieved through hardware, software portability becomes more important than software efficiency.

f. Reliability

Reliability refers to the ability of the software to be used in a production setting. Software unreliability has a number of sources:

- the target hardware not meeting its reliability specifications
- system complexity that creates the likelihood that schedules cannot be met
- unreliable supporting software in the development environment (e.g., compilers, debugging tools, trace tools)

These sources of risk should be addressed during the design phase and are usually consigned to the risk management agenda of the project.

3. Implementation Risks

This element addresses the sources of project risk associated with how the coding will be done, that is, how the design will be translated into unit specifications and ultimately "units" of code. Attributes of this element describe the nature of risks associated with the quality and stability of software or interface specifications, and coding constraints or even "styles" that, if left unspecified, may exacerbate future maintenance and extensibility problems. Software specifications are as important to the project success as are the more obvious higher level "requirements" explored earlier. Project teams dominated by physical or natural scientists or engineers may not recognize this, which often exacerbates the impacts of these risks.

a. Specifications

Specifications are typically the output of the design step; they describe how the requirements are to be met in the code to be developed and drive the planning process. Requirements do not generally define an application code; the specification documentation (detailed design) should. Therefore, vetting the specifications back to the sponsor/customer and ahead to the implementers is important, as mentioned above. Module interfaces, for example, are beyond the scope of most requirements (sponsors have little reason to care), but should be fully spelled out in the specifications. Specifications may be inadequate in at least the following ways:

- they may not accurately reflect the requirements
- they may not be complete or detailed enough to guide the development activity

Specifications should also provide the basis for the test plan; often the adequacy of testing is judged against the specification, not the requirements.

Another source of risk associate with the specifications is the lack of coherence, especially with regard to third-party components.

b. Project Plan

A project plan translates the specifications into a plan of action with a schedule, resources, and budget. In the scientific software environment, the project plan must be flexible and adaptable to address the changing needs and goals of the project—especially those with research objectives or unknown requirements. It is often true that the achievement of a project milestone has the potential to change the direction of the project. Moreover, rapidly evolving software, hardware, and scientific technologies also have this potential. ("Rapidly evolving" refers to changes that occur over shorter time scales than the code development project itself.)

Nevertheless, to project stakeholders, the project plan may be the only tangible project asset in the initial phases. To the extent that there is no plan or one that does not adequately keep the stakeholders in the project (i.e., upper management, sponsors, customers) informed, there is a risk that the project may face problems with continued support. Of course, the execution is also at risk.

c. Scale of Effort

The technical challenges presented by the sheer scale of complex applications development typical of the HPC environment is a source of risks related to

- the ability to satisfy hardware performance expectations, especially the scaling of performance with more resources such as processors
- the ability to utilize system resources (e.g., distributed memory, caches, and threading)
- the complexity of system integration involving multiple programming languages and fragile operating systems
- the difficulty of debugging in an HPC environment

Also, as scientific/engineering software projects grow more complex, the ability of an individual team member to grasp the consequences of individual actions on the unfamiliar parts of the project is a source of risk. This is especially true in the "system of systems" case, that is, when the code under development is a part of a much larger system whose other components are developed by other teams.

In the Constructive Cost Model (COCOMO) estimation models, "scale" is the most important factor contributing to a project's (or, in the case of a system of systems, a major component's) duration and cost. Consequently, scale must be viewed as a potential source of risk [Boehm 1995].

4. Testing and Evaluation Risks

In the original [Carr 1993], "testability" risks were an attribute of design risk, not a class by itself. Owing to the importance of the verification and validation of scientific/engineering software, it has been included as a source of risk at a higher level in this taxonomy. Just as scientific/engineering software must have requirements and a design codified into a specification document, it must also have a documented test plan. Most testing will address specifications, but important validation (in part B, below) will address the requirements. All test plans should include a test coverage matrix documenting just what is being tested and how. The main consequence of the sources of risk cited below is that it will not be possible to demonstrate that the code is actually fit to purpose.

a. Verification

Verification refers to ensuring that the code solves the equations of the model correctly. Since scientific/engineering software cannot be exhaustively tested through a deterministic, combinatorial approach, there is always a risk that some feature or aspect of it cannot be verified. Project planning therefore demands the inclusion of better and perhaps more statistically sophisticated test methods, often requiring the expertise of a group (typically statisticians) distinct from the development team, to minimize the risk that the code does not implement the algorithms of the model correctly. Another good reason to use an independent test organization is that they are less prone to share unrecognized assumptions. Verification is typically conducted at different stages of development:

i. Unit Testing

Unit testing refers to the testing of the basic features of the code, usually found in individual modules or "units." (A software "unit" is a component that is not subdivided into other components [IEEE 1990]. Risk factors affecting unit testing include:

- availability of pre-planned test cases that have been verified to test unit requirements
- availability of a test bed consisting of the necessary hardware or emulators, and software or simulators
- availability of test data to satisfy the planned test
- sufficient schedule to plan and carry out the test plan

ii. Integration Testing

Integration testing refers to the testing of multiple units of the code as an ensemble. The risk factors cited above for unit testing also apply here; however, the first one, the availability of pre-planned cases, is often a larger issue and a more significant source of risk.

iii. Interoperability Testing

Here the scale of testing advances to include the interaction with the target hardware, if that has not already occurred, as well as the interfaces to third-party applications that users, not developers, employ (like visualization tools), and other aspects of the actual user environment.

b. Validation

Validation refers to determining whether the mathematical model instantiated in the code faithfully mimics the intended physical behavior. This is a central difficulty for scientific/engineering codes because, in many instances, the software is developed to model behavior that is too expensive, too dangerous, or impossible to test in the first place. Another manifestation of the risk associated with validation is that the region of applicability of the underlying model upon which the code is based may not be known. These risks should be quantified up front as accurately as possible to ensure that unrealizable expectations are avoided. Finally, there is an increasing possibility that the code is part of a larger system, each component of which carries the risks described above in unknown proportions. Validating the components of a system is not the same as validating the whole system. In a sense the users of scientific codes are the final arbiters of the domain of applicability of these codes, but there is a risk that their determinations will not benefit from any knowledge of the assumptions of the developers. It can be argued that one of the goals of validation testing should be to determine the region of applicability of the code. Validation testing can usually only be done at the end of the development process.

B. Development Environment Risks

The "development environment risk" class addresses the sources of risk inherent in an environment and the processes used to develop a software application. The risks here are usually intrinsic, but in some instances the choice of development environment, or some of its features, is beyond the control of the code development team. This environment includes the development philosophy, (e.g., Capability Maturity Model [CMM], agile), workflow management model (e.g., incremental, iterative, spiral, and others), the development system, project management methods, and work environment. The risk elements associated with the development environment are characterized below.

1. Development Process Risks

This element refers to risks that can be experienced through a process or processes by which the development team proposes to satisfy the customer's requirements. The process is the sequence of steps leading from the initial requirements gathering and specification to the final delivered software product. Development processes themselves have attributes. Most conform to some degree to a *development philosophy* like CMM, ISO, or agile. Most development processes can also be identified with a *workflow management model* (called "development models" in the original SEI risk taxonomy).

The development philosophy typically describes the approach to processes used to create a software product [Paulk 1993]. Examples include formal methods favored by CMM or ISO, and agile methods, which are often encountered in scientific/engineering code development projects. CMM-endorsed processes emphasize a formal approach to the customary development phases (i.e., life-cycle elements [IEEE 1990]) of requirements analysis, product design, product creation, testing, delivery, and maintenance (sometimes called "production," ending ultimately in the eventual retirement or decommissioning of the application). It includes both general management processes such as costing, schedule tracking, and personnel assignment, and also project-specific processes such as feasibility studies, design reviews, and regression testing. Importantly, advanced CMM organizations collect and utilize metrics about their own development processes with a view to process improvements. Agile methods, on the other hand, focus at a philosophical level on software development *practices and people*, not *processes*. Note that some agile methods are very prescriptive; the reference above is intended to capture the shared philosophical basis of this development methodology [Agile Software]. The lack of a methodology has been recognized as a risk in and of itself. Specific sources of risk associated with the absence of a development methodology include the absence of coherent change control, no project planning, and no repeatable processes or practices. Of course, the adoption of—or more likely, the imposition of—a methodology incompatible with the goals of the project or the development team is also a source of risk.

Workflow management models describe different approaches to the management and organization of the development project workflow elements cited above [Beck 1999]. Various models have been proposed for this: waterfall (the original conceptual model for software development),

incremental, iterative, evolutionary, spiral (emphasizing prototyping), and others. At the opposite end of the spectrum from the waterfall model are approaches like extreme programming (XP) and rapid application development (RAD). Note that the software engineering literature often aligns these workflow management models with certain development methodologies, e.g., RAD with agile.

This element groups risks that result from a development philosophy and/or workflow management approach that

- does not reflect what is known at the beginning of the project
- is not suited to the activities necessary to accomplish the project goals
- is poorly communicated to the project staff and lacks enforceability

a. Repeatability

Product quality is strongly correlated with development process repeatability. Whether repeatability is associated with formal development processes or rigorously followed practices, it is cited by all of the major development philosophies as a requirement for success. The lack of repeatability may result in the inability of the team to reconstruct, extend, or even modify in a deliberate way its approach.

b. Suitability

Suitability refers to the support for the type and scope of the activities required for a specific software development project provided by the selected development philosophy, process, methods, and tools. For example, the adoption of too much formality may put agility at risk in projects where flexibility is important.

c. Control of Process

Some form of process control is usually necessary to ensure that

- the processes or practices adopted by the project are adhered to
- monitoring of quality and productivity goals occurs

Control may be complicated when development occurs at distributed sites. Control of process is an important aspect of scheduling. Two significant control-related sources of risk here are (1) artificial deadlines, that is, deadlines not reflected in the project plan derived from the specifications, and (2) preemption of the schedule by a project with higher priority. The main consequence of a lack of control of process is that the software project management is not able to assess the state of the project at any point in time (the “lost in the woods” syndrome).

d. Familiarity with Process or Practice

Lack of familiarity with the development process or practices often results in the failure of the development team to adopt them. Scientific/engineering code developers tend to be process-averse. Most are scientists or engineers, not software engineers or even professional programmers. A special effort is usually necessary to get such teams to follow development processes, in par-

ticular. Success is greatest when the teams recognize the value and benefits of the process or practice.

e. Environment Change Control

The development environment itself will inevitably not be static during long software development cycles. Some orderly way to adapt to change is necessary, but is often overlooked by scientific code developers. Unanticipated and unplanned changes in the development environment—even something as simple as a compiler upgrade—can disrupt the development schedule.

2. Development System Risks

The development system risk element addresses those risks related to the choices of hardware and software tools used in application development. The purpose of these tools is to facilitate application development and in some cases (such as integrated development environments) to reduce performance risk as well (for example, by introducing automated product control).

a. Hardware Capacity

Inadequate capacity of the development system may result from too few development workstations, insufficient access to the target platform, or other inadequacies in equipment to support the scheduled pursuit of unit development, code integration, tuning, and testing activities. One or another of these sources of risk exists for almost all projects at some point in their lives. The consequence is project delay.

b. Development System Capability

This attribute refers primarily to the completeness and maturity of the tools of the trade for the scientific/engineering software developer:

- compilers
- linkers
- schedulers
- debuggers
- memory checkers
- profiling toolkits and APIs
- tracers
- visualizers
- bug tracking tools
- integrated development environments
- mathematical program libraries
- memory management libraries

Secondarily, it refers to the capabilities of the hardware used for development, including the target hardware. The performance, availability and long-term support of development tools is often (correctly) considered to be a major source of risk by scientific/engineering software developers.

c. Suitability

A development system that does not support the specific development models, processes, practices, methods, procedures, and activities required and selected for application development is a source of code team performance risk. This attribute also includes risks introduced by the management, documentation, and configuration management processes. In HPC environments there are often gaps in the availability of tools to support these activities. The closer the project is to the cutting edge of hardware evolution, the greater the likelihood that the software development environment will exhibit deficiencies that may not have been foreseen during scheduling.

d. Usability

Usability usually manifests in the presence of development system documentation, the accessibility of the system itself, and the ease of use of the features of the system. It is not uncommon for the software environments of state-of-the-art HPC machines to be fragile (e.g., unstable compilers or even operating systems), which limits the usability of the machines well into the development cycle. Unusable or difficult-to-use systems endanger project schedules and budgets in ways that may be difficult to quantify in advance.

e. Familiarity

Development system familiarity is a function of the prior use of the system by the host organization and by project personnel, as well as adequate training for new developers. Software developers for state-of-the-art HPC machines, no matter how experienced, often face steep learning curves when confronted with new languages, compilers, and computer architectures. It can be difficult to predict the impact of this learning curve on code team productivity. The lack of familiarity usually manifests in unreliable project development schedules and budgets.

f. Reliability

Here reliability means dependability, not usability. An unreliable development system is a major source of risk to code team productivity.

g. Target-Unique System Support

System support, including training and access to expert users and prompt resolution of problems by vendors, is crucial in the HPC environment. Lack of it can stop a development project in its tracks.

h. Security

Dealing with access security, especially in secure environments, usually adds steps to the development process and typically an underestimation of the effort required to address them. The long times required to obtain security clearances for capable and ready project staff is a common costly example.

3. Management Process Risks

This is the category of risks associated with planning, monitoring, and controlling budget and schedule; controlling factors involved in defining, implementing, and testing the software application; managing project personnel; and handling external organizations, including the customer, senior management, matrix management, and other contractors. It is widely recognized that management actions determine, and management is ultimately responsibility for, much of the risk associated with software development projects. Management processes must support the following central objectives [DeMarco 1999]:

- recruit the right staff
- match them to the right tasks
- keep them motivated
- help teams jell

Moreover, management commitment has been cited as the number *one* risk to long term project success [Schmidt 2001, Keil 1998].

a. Contingency Planning

The existence of a well-defined plan that is responsive to contingencies as well as long-range goals is necessary for the proper management of project resources, schedule, and budget. The plan must be formulated with the input or at least acquiescence of those affected by it. Not doing so has led to many software development failures.

b. Project Organization

The goal of project organization and management is to foster the creation and nurturing of a well-functioning team. In a poorly organized team the roles and responsibilities are not understood or followed. The importance of this attribute increases in proportion to the size and scope of the project. One project organization scheme does not fit all situations. The most egregious examples of faulty organization usually come from large, very informal, process-averse teams.

c. Management Experience

A lack of management experience can impede effective communication and decision making in software projects, and can manifest at all levels regarding

- general management
- software development management
- the application domain
- scale and complexity of the project and targeted hardware system(s)
- selected development process or practices
- development environment

d. Program Interfaces

Ineffective interactions have a negative impact on decision making and can occur among managers at all levels with program personnel at all levels, and with external personnel such as the customer, senior management, and peer managers.

e. Reward Systems

Reward systems are often a tacit component of the management process. Scientific/engineering software development projects are usually meritocracies. Consequently, if the rewards system is not perceived to be aligned with merit, there can be a negative impact on project morale.

4. Management Methods Risks

This element refers to the risks associated with methods adopted for managing both the development of the product and program personnel. These include risks related to quality assurance, configuration management, staff development with respect to program needs, and maintaining communication about program status and needs. The continuity of management support over the life of the project is an important facet of this element. Continuity is especially challenging in view of the fact that many important scientific code development projects have a production phase that spans careers—that is, decades long.

a. Monitoring

It is impossible to address problems if there is no mechanism in place to detect them. The main consequence of risk associated with unmonitored projects is that they will exhibit unpredictable and unexpected outcomes.

b. Personnel Management (Staffing and Training)

Personnel management generally refers to selection and training of program members to ensure that they

- take part in planning and customer interaction for their areas of responsibility
- work according to plan or at least expectations
- receive the help they need or ask for to carry out their responsibilities

Poor team performance may stem from failures to address personnel management issues. Sources of staffing risks include

- insufficient/inappropriate project staff
- staff volatility
- unnecessary use of outside consultants
- lack of domain and programming expertise

c. Quality Assurance

The term “software quality” typically refers to code that is as defect-free as is practical, and is maintainable, portable, and well-written (transparent). In the typical scientific/engineering soft-

ware development project, these attributes are not customarily specified contractually, but are expected deliverables of the development team. This does not necessarily happen spontaneously. Without management attention, there is the possibility that quality assurance will be either lacking or uncertain.

Another aspect of quality assurance is related to the concept of validation—can the code be validated as consistent with the laws of nature that it is intended to model. The consequence here is that even the perfect specimen satisfying the definition of the previous paragraph is of unknown value.

d. Configuration Management

For scientific/engineering codes the consequences of faulty configuration management (of both code and documentation) grow in proportion to

- the age of the code
- the size of the code
- the size of the code team
- the failure to use repeatable processes
- the failure to enforce standards

The main consequence of faulty configuration management is an unmanageable program library.

5. Work Environment Risks

This element refers to risks arising from subjective aspects of the environment such as the amount of care given to ensuring that stakeholders, including the management, users, sponsors, and the development team itself, are kept informed of program goals and information, the way they work together, their responsiveness to staff inputs, and the attitude and morale of the program personnel. A well-functioning development team has already been identified as a critical success factor for software development projects, scientific/engineering or otherwise.

a. Quality Attitude

It is important to recognize that scientists and engineers are usually more concerned with developing code that supports their scientific goals than with code that conforms to the customary notions of IT software quality. As far as the science is concerned, the code may be excellent, but it may be lacking in quality attributes important to sponsors and users. Nevertheless, it is the drive, focus, and scientific integrity of scientific code developers that is the source of quality in scientific/engineering applications. Misunderstandings about this are a constant source of tension between sponsors, managers, and code teams; education about the importance of sound software engineering process and practice continues to be needed.

b. Cooperation

Poor team relationships, which engender a lack of cooperation, can destroy code development projects. These may result from factors such as conflicting egos or even burnout. Management may not identify problems here soon enough to avoid damage to the project.

c. Communication

The goal of management communication is to ensure that knowledge of the mission, goals, requirements, and design goals and methods of the project are broadly understood by all of the stakeholders, specifically including the development team itself. Ineffective communications usually result in a misalignment between the goals of the stakeholders.

d. Morale

Morale has a strong impact on enthusiasm, performance, productivity and creativity. At the extreme end of consequences is anger that may result in intentional damage to the project or the product, an exodus of staff from the project, and harm to the reputation of the project organization that makes it difficult to recruit.

e. Trust

Trust is an attribute that is often taken for granted in the work environment. The consequence of a lack of trust is that all of the preceding attributes—cooperation, communication, morale, and even a quality attitude—will be diminished with an accompanying impact on project deliverables.

C. Programmatic Risks

Programmatic risks refer to those project risks emanating from "external" forces acting on scientific/engineering software development projects. These are sources of risk that are usually outside the direct control of the code development team, that is, extrinsic risks. From the development team's point-of-view, these risks are often considered "acts of God."

1. Resources Risks

This element addresses sources of project risk arising from resource dependencies or constraints that the project must honor. These dependencies/constraints include schedule, staff, budget, and facilities.

a. Schedule

The stability (and in some cases, feasibility) of the project schedule in the presence of changing internal and external events or dependencies—as well as the validity of estimates and planning for all phases and aspects of the project—is a source of risk that almost all software development projects face. Tight schedules and rigid deadlines are usually incompatible with scientific/engineering software development projects, which rarely can be planned at a fine level of granularity. Experience has shown that when schedules and milestones are dictated, projects are likely to fail.

b. Staff

The availability of project staff with adequate skills and experience is a prerequisite for success. Commitments to milestones and schedules are at risk if the project staff is deficient in numbers or skills and experience.

c. Budget

Like the schedule, the budget is a well-recognized [Boehm 1991] source of project risk. This tends to be a greater risk for software projects that extend over many budget cycles. Many long-lived scientific/engineering development projects are tied to annual budget cycles that force the code teams to manufacture artificial deliverables to ensure continued funding. There is an inviolate relationship between requirements, specifications, budget and schedule: imposing all of them a priori—absent design, specification and agreement from the implementers—is a recipe for failure.

d. Facilities

The availability of adequate project facilities, including computer and software support for development, integration, and testing of the application can impact project schedules. This is often overlooked in the project design and planning phases.

e. Management Commitment

A lack of management commitment has been cited as the *greatest* risk to software development projects of all stripes [Schmidt 2001, Keil 1998]. For many scientific/engineering software development projects in start-up mode, the first significant deliverable may not be available for three to six years from the start of the project. The managers who approved the project may well have moved on to other jobs, forcing the project team to remarket itself to new managers. A lack of management commitment is often fatal to a software development project.

2. Contract Risks

Risks associated with the program contract are classified according to contract type, restrictions, and dependencies.

a. Contract Type

Scientific/engineering code projects are often governed by level-of-effort agreements, cost plus award fee, cost plus fixed fee, or research grants. Any contract agreement that does not recognize the inherent difficulty associated with rigorous specification of the requirements increases the risk that contractual expectations will not be met. For example, cost and schedule are often estimated based on the history of previous similar projects. The contract vehicle must recognize the inherent uncertainties in this approach. Contract elements like the statement of work, data collection, testing requirements, and the amount and conditions of customer involvement are all subject to misinterpretation and are, therefore, potential sources of risk.

b. Restrictions

Contractual directives to use specific development methods, third-party software, or equipment may introduce uncertainties that cannot be addressed without evaluation. If development starts without this step, the project may founder.

c. Dependencies

Contractual dependencies on outside contractors or vendors, customer-furnished equipment or software, or other outside products and services are well-recognized sources of software development risk.

3. Program Interface Risks

This element consists of the various interfaces with entities and organizations outside the development program itself.

a. Customer Communication

Difficult working relationships or poor mechanisms for attaining customer agreement and approvals, not having access to certain customer factions, or not being able to communicate with the customer in a forthright manner are all sources of risk to the continued funding and ultimate acceptance of the software application.

b. User Commitment

Customers and users are not synonymous. Failure to gain user commitment and to manage user expectations has been cited as one of the top five threats to the success of all software development projects [Keil 1998].

c. Sponsor Alignment

In many cases, the customer, end user, and sponsor of a scientific/engineering software development project may not be the same, as is the case with most federally funded projects. The misalignment of the goals and expectations of the sponsor(s) with those of the customer(s) or end users is a source of project risk.

d. Subcontractor Alignment

Subcontractor alignment risks refer to those risks that arise from inadequate task definitions and subcontractor management mechanisms, or the failure to transfer subcontractor technology and knowledge to the program or host organization. Failure to transfer key technology from subcontractors places the sustainability of the code at risk.

e. Prime Contractor

When the project is a subcontract, performance risks may arise from poorly defined task definitions, complex reporting arrangements, or dependencies on technical or programmatic information.

f. Corporate Communication

Risks in the corporate management arena include poor communication and direction from senior management as well as non-optimum levels of support. Long-term projects are often buffeted by changes in the senior management of the host organization.

g. Vendor Performance

Vendor performance risks may present themselves in the form of unanticipated dependencies on deliveries and support for critical system components.

h. Political

Political risks may accrue from relationships with the company, customer, associate contractors, or subcontractors, and may affect technical decisions, schedules, and even support for the project.

References

[ACM]
ACM Taxonomy, www.computer.org/portal/site/ieeecs.

[Agile Software]
Manifesto for Agile Software Development, http://agilemanifesto.org.

[Beck 1999]
Beck, K. "Embracing Changes with Extreme Programming," *Computer*/IEEE, vol. 32, no. 10, 1999, pp. 70-77.

[Boehm 1991]
Boehm, B. W. "Software Risk Management: Principles and Practices," *Software*/IEEE, vol. 8, no. 1, 1991, pp. 32-41.

[Boehm 1995]
Boehm, B, et al. "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, no. 1, 1995, pp. 57-94.

[Carr 1993]
Carr, Marvin J., et al. "Taxonomy-Based Risk Identification," Technical Report CMU/SEI-93-TR-006, June 1993. http://www.sei.cmu.edu/publications/documents/93.reports/93.tr.006.html.

[DeMarco 1999]
DeMarco, T. & Lister, T. *Peopleware, Productive Projects and Teams*. Dorset House, New York, 1999.

[DeMarco 2003]
DeMarco, T. & Lister, T. *Waltzing with Bears: Managing Risks in Software Projects*. Dorset House, New York, 2003.

[IEEE 1990]
IEEE Standard Computer Dictionary, "A Compilation of IEEE Standard Computer Glossaries," IEEE Computer Society, IEEE-STD-610, ISBN 1-55937-079-3, 1990.

[Keil 1998]
Keil, Mark, et al. "A Framework for Identifying Software Project Risks," *Communications of the ACM*, vol. 41, no. 11, 1998, pp. 76-83.

[Paulk 1993]
Reference to "methodologies" in Cockburn, A. and Highsmith, J. "Agile Software Development, The People Factor;" *Computer* /IEEE, vol. 34, no. 11, pp 131-133. Also see Capability Maturity Model, v1.1, Paulk, M.C. et al. *Software*/IEEE, vol. 10, no.4, 1993, pp. 18-27.

[Process Impact]
See http://www.processimpact.com/articles/metrics_primer.html.

[Schmidt 2001]
Schmidt, R. et al. “Identifying Project Risks: An International Delphi Study,” *Journal of Management Information Systems*, vol. 17, no. 4, 2001, pp. 5-36.

[Weinberg 1998]
Weinberg, G. M. *The Psychology of Computer Programming*. Dorset House, New York, Silver edition, 1998.

Glossary

acceptance criteria
The criteria that a system or component must satisfy to be accepted by a user, customer, or other authorized entity. [IEEE-STD-610.12]

acceptance testing
Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system. [IEEE-STD-610.12]

application domain
Refers to the nature of the application. Here we are concerned with the high performance end of domain of scientific and engineering application.

audit
An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria. [IEEE-STD-610.12]

availability
The degree to which a system or component is operational and accessible when required for use. Usually expressed as the ratio of time available for use to some total time period or as specific hours of operation. [IEEE-STD-610.12]

baseline
A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures. [IEEE-STD-610.12]

baseline management
In configuration management, the application of technical and administrative direction to designate the documents and changes to those documents that formally identify and establish baselines at specific times during the life cycle of a configuration item. [IEEE-STD-610.12]

benchmark
A standard against which measurements or comparisons can be made. [IEEE- STD-610.12]

change control
A part of configuration management that reviews, approves, and tracks progress of alterations in the configuration of a configuration item delivered, to be delivered, or under formal development, after formal establishment of its configuration identification [IEEE-STD-610.12].

configuration
In configuration management, the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product [IEEE-STD-610.12].

configuration management
A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a controlled item, control changes to a

configuration item and its documentation, and record and report change processing and implementation status [IEEE-STD-610.12].

configuration management function
The organizational element charged with configuration management.

configuration management system
The processes, procedures, and tools used by the development organization to accomplish configuration management.

critical design review *(*CDR*)*
(1) A review conducted to verify that the detailed design of one or more configuration items satisfy specified requirements; to establish the compatibility among the configuration items and other items of equipment, facilities, software, and personnel; to assess risk areas for each configuration item; and, as applicable, to assess the results of producibility analyses, review preliminary hardware product specifications, evaluate preliminary test planning, and evaluate the adequacy of preliminary operation and support documents. See also: preliminary design review; system design review. (2) A review as in (1) of any hardware or software component. [IEEE-STD-610.12]

customer
The person or organization receiving a product or service. There may be many different customers for individual organizations within a program structure. Government program offices may view the customer as the user organization for which they are managing the project. Contractors may view the program office as well as the user organization as customers.

design specifications
A document that prescribes the form, parts, and details of the product according to a plan (also see design description. [IEEE-STD-610.12]

detailed design
(1) The process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented. See also: software development process. (2) The result of the process in (1). [IEEE-STD-610.12]

development computer
The hardware and supporting software system used for software development.

development facilities
The office space, furnishings, and equipment that support the development staff.

development model (also workflow management model)
The abstract visualization of how the software development functions (such as requirements definition, design, code, test, and implementation) are organized. Typical models are the waterfall model, the iterative model, and the spiral model.

development process
The implemented process for managing the development of the deliverable product. For software, the development process includes the following major activities: translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use. These activities may overlap and may be applied iteratively or recursively.

development sites
The locations at which development work is being conducted.

development system
The hardware and software tools and supporting equipment that will be used in product development including such items as computer-aided software engineering (CASE) tools, compilers, configuration management systems, and the like.

external dependencies
Any deliverables from other organizations that are critical to a product's success.

external interfaces
The points where the software system under development interacts with other systems, sites, or people.

hardware specifications
A document that prescribes the functions, materials, dimensions, and quality that a hardware item must meet.

implementation
The process of translating a design into software. [IEEE-STD-610.12]

integration
The process of combining software components, hardware components, or both, into an overall system. [IEEE-STD-610.12]

integration environment
The hardware, software, and supporting tools that will be used to support product integration.

integration testing
Testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them. See also: component testing; interface testing; system testing; unit testing. [IEEE-STD-610.12]

internal interfaces
The points where the software system under development interacts with other components of the system under development.

maintainability
The ease with which a software system can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. [IEEE-STD-D610.12]

modularity
The degree to which a computer program is composed of discrete components such that a change to one component has a minimal impact on other components. [IEEE-STD-610.12]

portability
The ease with which a system or component can be transferred from one hardware or software environment to another. [IEEE-STD-610.12]

preliminary design
The process of analyzing design alternatives and defining the architecture, components, interfaces, and timing and sizing estimates for a system or component. See also: detailed design.

procedure
A written description of a course of action to be taken to perform a given task. [IEEE-STD-610.12]

process
A sequence of steps performed for a given purpose; for example, the software development process. [IEEE-STD-610.12]

process control (as in management)
The direction, control and coordination of work performed to develop a product. [IEEE-STD-610.12]

product integration
The act of assembling individual hardware and software components into a functional whole.

quality assurance
A planned and systematic pattern of actions necessary to provide adequate confidence that a product conforms to established technical requirements. [IEEE-STD-610.12]

reliability
The ability of a system or component to perform its required functions under stated conditions for a specified period of time. Usually expressed as the mean time to failure. [IEEE-STD-610.12]

requirements analysis
(1) The process of studying user needs to arrive at a definition of system, hardware, or software requirements. (2) The process of studying and refining system, hardware, or software requirements. [IEEE-STD-610.12]

reuse
Hardware or software developed in response to the requirements of one application that can be used, in whole or in part, to satisfy the requirements of another application.

safety
The degree to which the software product minimizes the potential for hazardous conditions during its operational mission.

security
The degree to which a software product is safe from unauthorized use.

software architecture
The organizational structure of the software or module.

software life cycle
The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. [IEEE-STD-610.12]

software requirement
A condition or capability that must be met by software needed by a user to solve a problem or achieve an objective. [IEEE-STD-610.12]

software requirements specification (SRS)
Documentation of the essential requirements (functions, performance, design constraints, and attributes) of the software and its external interfaces. [IEEE-STD-610.12]

system integration
The act of assembling hardware and/or software components into a deliverable product.

system requirement
A condition or capability that must be met or possessed by a system or system component to satisfy a condition or capability needed by a user to solve a problem.

system testing
Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. See also: component testing; integration testing; interface testing; unit testing. [IEEE-STD-610.12]

target computer (machine)
The computer on which a program is intended to execute. [IEEE-STD-610.12]

test specifications
A document that prescribes the process and procedures to be used to verify that a product meets its requirements (sometimes referred to as a test plan). [IEEE-STD-610.12]

traceability
The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another. [IEEE-STD-610.12]

unit
(1) A separately testable element specified in the design of a computer software component. (2) A logically separable part of a computer program. (3) A software component that is not subdivided into other components. [IEEE-STD-610.12]

unit testing
Testing of individual hardware or software units or groups of related units. See also: component testing; integration testing; interface testing; system testing. [IEEE-STD-610.12]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	January 2007	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
A Proposed Taxonomy for Software Development Risks for High-Performance Computing (HPC) Scientific/Engineering Applications	FA8721-05-C-0003

6. AUTHOR(S)

Richard P. Kendall; Douglass E. Post; Jeffrey C. Carver; Dale B. Henderson; & David A. Fisher

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2006-TN-039

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Because the development of large-scale scientific/engineering application codes is an often difficult, complicated, and sometimes uncertain process, success depends on identifying and managing risk. One of the drivers of the evolution of software engineering, as a discipline, has been the desire to identify reliable, quantifiable ways to manage software development risks. The taxonomy that follows represents an attempt to organize the sources of software development risk for scientific/engineering applications around three principal aspects of the software development activity: the software development cycle, the development environment, and the programmatic environment. These taxonomic classes are divided into *elements* and each element is further characterized by its *attributes*.

14. SUBJECT TERMS	15. NUMBER OF PAGES
managing risk, software development risks, taxonomy, software development activity, software development cycle, development environment, programmatic environment	39

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL